Une introduction aux QObject
Application au binding dans QML

1- Table des matières

1- Généralités sur les Q_Object

Les **QObject** sont centraux dans l'utilisation de **Qt**, en particulier lors de la création d'IHM en QML.

Ils reposent sur des macros.

Ils possèdent aussi la particularité de pouvoir communiquer entre eux grâce à des méthodes : les **signaux** et des **slots**.

Des objets ainsi **connectés** (même rattachés à des threads différents, hors programme) peuvent envoyer et recevoir des informations.

On soulignera surtout ici l'intérêt de ces objets pour l'utilisation de QML.

Création d'un QObject

Un **QObject** nécessite un #include <QObject>.

Il doit être définit dans un projet **Qt**, c'est-à-dire avec un main qui lance QGuiApplication ou QCoreApplication, etc….

Les **QObject** sont en général utilisés en dérivation (ie en créant un objet héritant d'un **QObject**).

Nombreux objets de la bibliothèque **Qt** dérivent de **QObject**, et on peut créer nos propres classes héritant de **QObject**.

Le **constructeur de recopie** et **l'opérateur d'affectation** sont désactivés par défaut, on codera des fonctions qui recopieront les valeurs si nécessaire.

Lien

Vous trouverez les dernières informations concernant les QObjects

Casts de pointeurs sur des QObject

Comme pour tous les types de pointeurs, il est possible d'effectuer des **casts dynamiques** d'objets héritants de **QObject**, vers la classe dérivée.

Cette opération permet de passer d'un type parent à un type enfant.

Il est toujours possible d'utiliser `qobject_cast` sur un pointeur, on vérifie ensuite dynamiquement la validité du résultat avec un **test conditionnel** (`if{ …`).

La syntaxe est la suivante :

```
TypeDérivé* = qobject_cast< TypeDérivé *> (instance_objet_à_caster);
```

Par exemple dans le QML_CRUD on écrira

```
Personne* p = qobject_cast<Personne *> (op);
if( p )
{
    //traitement
}
```

Pour pouvoir utiliser les méthodes et propriétés de l'objet hérité, du coté C++.

On notera le **test conditionnel** qui permet de vérifier que le **cast** s'est bien passé, ce qui doit-être le cas sauf erreur de conception.

Comme pour tout pointeur on n'oubliera pas de faire une **allocation** mémoire à chaque création de pointeur sur objet, et sa **désallocation** quand nécessaire.

Retour sur les fichiers moc

Les QObject utilisent des **macro** particulières : **Q_OBJECT**, mais aussi **Q_PROPERTY** et **Q_INVOKABLE**…..

Grace à ces directives, lors de l'étape de la pré-compilation des fichiers C++ **intermédiaires** sont générés **automatiquement** avec une **extension** *.moc*.

Ces fichiers sont ensuite **compilés** comme d'autres fichiers C++, (ie seront utilisés lors de la génération de l'exécutable final).

Le mécanisme des ***signaux*** et des ***slots*** est mis en œuvre dans ces fichiers, ainsi que la gestion de la mémoire de tous les QObjects enfants (Cf. le paramètre parent d'un constructeur de QObject).

Ces fichiers ne doivent jamais être édités car ils sont re-générés à chaque compilation du projet.

Les fichiers et les slots étaient surtout utilisés lors de la création d'IHM avec des QWidget.
Ils continuent à être utilisés avec QML, comme détaillé dans les paragraphes sur le QML :

2- C++ et QML

Nous avons souvent répété qu'il est préférable de séparer les couches d'une application.

En particulier pour nos applications de clients lourds ou d'applications mobiles, on utilisera les couches suivantes :

Une couche QML ; une couche intermédiaire passerelle appelée Contexte (et appelée **Context** en QML) ; les Objets C++.

Un projet de référence QML_CRUD est fourni avec ce document.

Passage de C++ à QML

Tous les objets qui doivent être affichés en QML doivent hériter de **QObject**.

Un passage de valeurs de C++ à QML s'effectue avec l'instruction `setContextProperty`.

Pour simplifier l'architecture du projet, nous avons choisi dans notre projet de mettre tout le code correspondant à ces passages de valeurs dans une seule classe Contexte.

Cette classe gérera donc les passages de valeurs des instances d'objets d'une part et des listes d'instances d'autre part.

i. L'objet Contexte

L'objet Contexte (définit avec contexte.h et contexte.cpp) est **la passerelle** que nous avons définie vers le QML, grâce aux deux lignes suivantes du main.cpp :

```
    // Set le ctx
    contexte.setContext( ctx );

    // Update le ctx
    contexte.updateContext();
```

(On pourrait le définir en tant que singleton (ie. comme objet unique dans l'application))

Pour mémoire c'est la ligne suivante de la méthode **setContext(…)** qui permet l'utilisation en QML

```
m_context->setContextProperty("Context", this);
```

Et c'est donc pour cela qu'il n'y a pas de « e » dans le QML, (on pourrait utiliser une autre valeur) !

ii. Mise en pratique du **binding** avec un type natif Qt

Un type natif peut-être int, double,…. et un type Qt seront les QString, QPoint, QColor, etc…. qui peuvent être directement utilisé avec la macro **Q_PROPERTY**.

En écrivant la macro ci-dessous dans contexte.h :

```
Q_PROPERTY(QString chaine1 READ getChaine1 WRITE setChaine1 NOTIFY chaine1Changed)
```

Puis en cliquant droit dessus, ensuite *Refactor*, et *Generate Missing PROPERTY Members*, on peut générer l'attribut et les méthodes correspondantes :

```
    QString getChaine1() const
    {
        return m_chaine1;
    }

signals:
    void chaine1Changed(QString chaine1);

public slots:
    void  setChaine1(QString chaine1)
    {
        if (m_chaine1 == chaine1)
            return;

        m_chaine1 = chaine1;
        emit chaine1Changed(m_chaine1);
    }

private :
    QString m_chaine1;
```

On note les choses suivantes :

La variable nécessaire est automatiquement créée : elle est précédée de **m_** et est **privée**.

La fonction `getChaine1` est un **getter** (**manipulateur**) qui ne modifie pas l'objet.

La fonction `chaine1Changed` est un **signal**, on **admettra** qu'il permet donc de propager l'information dans tout le programme aux objets qui lui sont attachés.

La fonction `setChaine1` est un **slot**, qui utilise le **signal** ci-dessus.

C'est grâce à ce signal que les informations sont mises à jour.

Remarque :

`getChaine1` est souvent écrite en `chaine1`, il suffit de bien savoir à quoi correspond chaque nom dans Q_PROPERTY.

Evaluation QML

La valeur de la variable `chaine1` de Contexte pourra donc être **récupérée** en QML comme ci-dessous :

```
Text{
   text : Context.chaine1
}
```

C'est `getChaine1` qui sera invoquée au C++ à chaque évaluation de `chaine1` !

Modification

La valeur de `chaine1` pourra aussi être **modifiée** avec une instruction *javascript* comme ci-dessous :

```
    Button{
        //....
        onClicked: {
            Context.chaine1 = "ok"
        }
    }
```

C'est dans ce cas la méthode `setChaine1` de Contexte qui sera invoquée en C++ !

On remarque qu'à l'émission du signal `emit chaine1Changed` de `setChaine1`
la valeur du texte `Context.chaine1` est **réévaluée**, induisant l'appel de `getChaine1` par le moteur de QML autant de fois que nécessaire.

Exercice : Vérifier ce comportement par des traces dans le code.

On peut aussi vouloir afficher les attributs d'un **objet** :

iii. Instances d'objets uniques

Les **objets** qui seront affichés en QML devront tous hériter de **QObject**.

Pour accéder à notre objet en QML, on utilisera le même code que pour un type natif, sauf que le type sera un **pointeur sur notre objet** :

```
    Q_PROPERTY(Personne* personne READ personne WRITE setPersonne NOTIFY personneChanged)
```

On génèrera de la même manière les membres et propriétés pour notre instance **personne** (de la classe Personne), elle-même propriété de l'objet Contexte.

Attention à bien **allouer de la mémoire** au pointeur m_personne dans le constructeur de Contexte !

Les propriétés qui seront accessibles en QML devront utiliser les **Q_PROPERTY** comme défini plus haut.

Et on pourra utiliser le QML comme suit :

```
Text{
    text : Context.personne.nom
}
```

Il est important de se rappeler que ce sont les fonctions `personne` et `setPrenom` qui sont appelées, lors de l'affectation en QML :

```
Context.personne.prenom = "Johann Sebastian"
```

On peut aussi accéder à des **Q_Object**s membres, comme pour le Panier d'une Personne, et afficher une liste de valeurs avec des QStringList.
Cette possibilité est suffisante pour afficher des tables, etc….. comme dans le projet.

iv. Listes d'objets

Un conteneur `QList` qui contiendra la liste des objets à afficher sera également déclaré comme membre de notre classe Contexte.

Dans la pratique on utilisera souvent une `QList de QObject*` pour pouvoir afficher notre liste d'objets définie en C++ du coté QML dans un composant comme ListView, GridView ou TableView (grâce aux propriétés ***model*** et ***delegate*** de ces composants).

(La syntaxe de passage au QML est détaillée dans le paragraphe 'Déclaration de la connexion vers le QML')

On définira par exemple,

```
QList<QObject*> m_listePersonnes;
```

On pourra mettre un pointeur nouvellement créé directement dans notre tableau :
le **cast** se fera **automatiquement** vers la classe parent `QObject*` compte tenu du type défini de `listePersonnes`.

```
m_listePersonnes.push_back( new Personne );
```

Nous aurons alors besoin du **cast** pour une réutilisation en C++ d'une instance contenue dans le tableau `listePersonnes`.

La classe Contexte donc gère directement la liste de personnes, il est possible pour le QML d'y accéder directement, avec la ligne suivante,

```
m_context->setContextProperty("modelePersonnes", QVariant::fromValue( m_listePersonnes );
```

La **listeView** peut afficher des données de chaque personne avec le model: *modelePersonnes*

Remarques :

- Le binding des listes d'objets n'est donc pas obtenu comme une propriété de notre objet Contexte, il est déclaré « directement ».

- Il est possible de gérer des propriétés comme attributs d'un **Q_Object** au lieu de le faire directement, avec QQmlListProperty. Nous avons choisi de ne pas le faire pour simplifier notre implémentation.

- Il nous est toujours possible de gérer des QStringList propriétés d'objet à la place pour afficher ces valeurs en QML, ce qui nous suffira amplement pour coder des applications.

- Il est également possible d'utiliser QAbstractListModel (Cf. *Using C++ Models with Qt Quick Views*)